Acta Cybernetica **21** (2013) 177–190.

x86 Instruction Reordering for Code Compression

Zsombor Paroczi*

Abstract

Runtime executable code compression is a method which uses standard data compression methods and binary machine code transformations to achieve smaller file size, yet maintaining the ability to execute the compressed file as a regular executable. With a disassembler, an almost perfect instructional and functional level disassembly can be generated. Using the structural information of the compiled machine code each function can be split into so called basic blocks.

In this work we show that reordering instructions within basic blocks using data flow constraints can improve code compression without changing the behavior of the code. We use two kinds of data affection (read, write) and 20 data types including registers: 8 basic x86 registers, 11 eflags, and memory data. Due to the complexity of the reordering, some simplification is required. Our solution is to search local optimum of the compression on the function level and then combine the results to get a suboptimal global result.

Using the reordering method better results can be achieved, namely the compression size gain for gzip can be as high as 1.24%, for lzma 0.68% on the tested executables.

Keywords: executable compression, instruction reordering, x86

1 Introduction

Computer programs, more precisely binary executables are the result of the source code compiling and linking process. The executables have a well defined format for each operating system (e.g. Windows exe and Mac binary [6, 15]) which usually consists of information headers, initialized and uninitialized data sections and machine code for a specific processor instruction set.

Compression of executables is mainly used to reduce bandwidth usage on transfer and storage needs in embedded devices [11]. Even today Linux kernels for embedded devices are stored in a compressed file (so called bzImage) [1], and every program for Android comes in a compressed format (apk) [20]. The literature distinguishes between two different approaches whether the produced compressed

*Department of Telecommunications and Media Informatics, Budapest University of Technology and Economics, E-mail: paroczi@tmit.bme.hu

DOI: 10.14232/actacyb.21.1.2013.13

binary remains executable without any additional software or hardware or not. In the first case the resulted binary includes the decompression algorithm and some bootstrapping instructions for restoring the original binary during runtime within the memory of the device. In the second case, the decompression is done on the operating system level or by special hardware, which has a predefined compression method with built in parameters. In this case the decompression method and the additional dictionaries is not an overhead in each binary, which allows decompression algorithms to use bigger dictionaries. If the decompression is hardware based, the performance can be significantly improved, which can also limit pure software based solutions.

Runtime executable code compression is a method which uses standard data compression methods and binary machine code transformations to achieve smaller file size, yet maintaining the ability to execute the compressed file as a regular executable. In our work we focus on machine code compression for Intel x86 instruction set with 32bit registers for both type of code compression approaches. It is important to note that the same method with minor modifications should work on each instruction set.

Various compression methods for the x86 machine code have been developed over the past years. Most of them use model based compression techniques (such as Huffmann coding, arithmetic coding, dictionary-based methods, predication by partial match and context tree weighting) with CPU instruction specific transformations such as jump instruction modification [2, 7]. These compression algorithms are lossless, so the binary before compression and after decompression is identical. In most cases the CPU instruction specific transformations are also reversible, the most common transformation is the jump instruction modification. x86 long jump instruction is 5 byte long. The first byte identifies the instruction, the last 4 is a relative jump address. The transformation modifies the jump address from relative to absolute before compression and does the inverse after decompression. This transformation may help producing smaller compressed binaries because most of the addresses used in x86 are absolute, so the transformed jump instructions have a better chance for model based matching and range encoding.

The compiled machine code is usually in one section of the executable, a continuous chunk of raw data, with a few known entry points. There is no clear indication, where each function or even instruction begins, it all depends on the actual execution. A function is a sequence of instructions in one continuous block with the following limitation: jump and call instructions from one function can transfer execution either into another part of the same function or to the first instruction of another function. If functions are called from outside, they always return and restore the stack according to the used call convention.

Very few compression methods try to identify functions. Most of them use binary pattern matching for known instructions - such as first byte match for jump transformation. When the executable is generated by a “standard” compiler certain patterns can help to identify functions in the code - such as stack protection stubs (push ebp; mov ebp, esp). Using a disassembler, an almost perfect instructional and functional level disassembly can be generated [21]. Using the structural information

of the compiled machine code, each function can be split into so called basic blocks, sequences of instructions including zero or one control transfer instruction at the end of the block [3, 9].

In our work we demonstrate that the reordering of instructions within a basic block can improve code compression without changing the behavior of the code. Our approach searches for local compression optimums on the function level for each function than combines these results to achive a suboptimal global result. With this novel approach we achieved the compression size gain for gzip can be as high as 1.24%, for lzma 0.68% on the tested executables.

2 Related work

Compression techniques for executables have been investigated by a number of researchers. Research directions can be categorized into two different groups: modification of the compiled (and sometimes not linked) machine code for smaller size or better compression, and different compression methods and models for the specific task of executable compression.

In the first category researchers tried to exploit the potentials of the same methods used in compilers, which resulted in aggressive inter-procedural optimalization of repeated code fragments, but without actual data compression methods. To increase the matching code segments the register renaming and local factoring transformation was used [11]. Using data dependency with instruction reordering was presented in the PLTO framework, which used the conservative analysis to gain performance with optimizing cache hit, instruction prefetching and branch predication [23]. Basic block reordering, without modifying the block itself, is also used to gain performance, mainly due to branch predication [18].

Reusing multiple instances of identical code fragments have been also exploited in the Squeeze++ framework, but it was done on various levels of granularity: procedures, code regions with unique entry and exit points, basic blocks and partially matched basic blocks [24, 8, 9].

Some works focus on the structural analysis of the whole program graph to identify common patterns [5, 10] or some other ways to simplify the structure of the graph [14, 5, 9] or even discover non-trivial equivalence in it [10, 13]. These methods focus on the control flow nature of the machine code, on the basic block level, they do not attempt to modify the code using the data flow information.

Refined compactation methods (unreachable code elimination, whole function abstraction, duplicate code elimination, loop unrolling, branch optimalization) can be used in various fields, which require specific tasks such as operation system kernel compactation [13] and ARM specific compactation [10].

For simplicity, all of these methods used additional link-time data to help separating the pointers (relocation data) from constant values, our method doesn't rely on such data. All of these compactation methods aim for smaller file size by modifying the machine code, yet maintaining the executability. But non of these methods exploit the results of the data compression research field, in fact some of these

compactations have a negative effect, if the machine code is later compressed [12].

Compression method evaluation and modification is also a widely researched field. Recompressing Java class files with various compression favorable modification resulted in significant gains, modifications include custom opcodes and string reordering, with the gzip compressor used as a black-box compressor [22]. Without further knowledge, in compiled machine code stream compressors, like gzip, are widely used as a black-box compressor, including android executables and linux kernels [5]. Compression can benefit from the known structure of the data, split-stream compression techniques conceptually split the input stream of instructions into separate streams, one for each type of instruction field. This lossless compression strategy can improve the compression rate significantly [19]. The same concept of separating the machine code into multiple context models can be used with instruction reordering, which can be compressed as non-sequential data [7]. In systems with limited memory, such as embedded systems, decompression methods have more limitations. Some algorithms decompress functions on demand, which means that the functions are decompressed on the first access [5].

All of these compression methods either use data compression as a black-box to simply compress the resulted binary as a stream, or use some special heuristic to supposedly encourage redundancy. This will always result in a suboptimal solution, neither of these works experiment with all the possible reorderings, due to the complexity. In our solution, the granuality of the search is fixed on the function level, but all the possible reorderings are tested, resulting the function level optimum of the compression.

The most closely related work to ours is PPMexe [12]. In this work the reordering is done globally (A1 algorithm) and on basic block level (A2, A3 algorithm), but the goal is to improve the predication rate of PPM by maximizing the number of n-symbol context occurrences. PPM highly depends on split-stream coding, a reversible transformation, which separates the structural information of an instruction into different streams. Our approach uses gzip and lzma and treats the machine code as one stream, searching for local optimum on the function level.

3 Reordering instructions

Original basic block	Reordered basic block
mov eax, ebx	**mov ecx, edx**
mov ecx, edx	**mov eax, ebx**
add eax,ecx	add eax,ecx
ret	ret

Table 1: Instruction reordering

Instructions have a predefined order in the executable, but in fact, every instruction could be executed once every required parameter is available. This statement is referred as data dependency. The basic idea behind reordering is that changing

the instruction order within a basic block will produce different raw data, which could lead to different compressed size, that may be smaller than the compressed original one. An example of reordering can be seen in Table 1. The original and the reordered basic block are functionally equal, after running both on various input the effect (observable registers and flags) are the same for both code sequence. The byte representation of the blocks in this example differ in 2 bytes. The original byte sequence is 6689d86689d16601c8c3 in hexadecimal notation while the reordered byte sequence is 6689d16689d86601c8c3, which can lead to better compression - for example if other parts of the code section has the pattern of d86601.

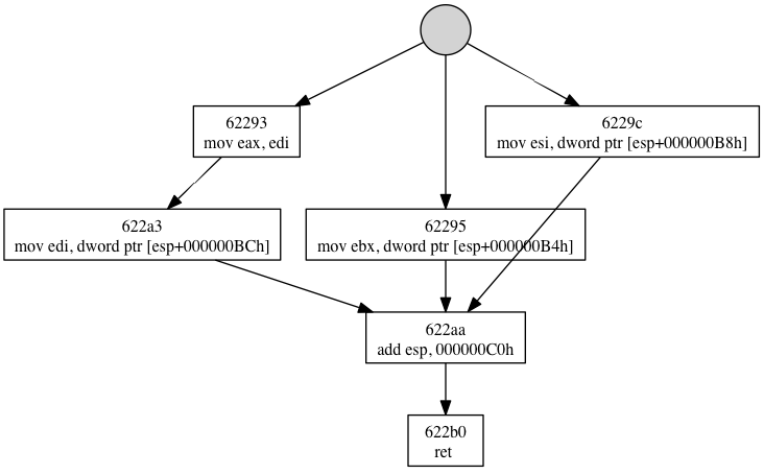

Figure 1: Data dependency graph inside a basic block

To ensure that the reordered block has exactly the same effect as the original one, the data dependency relations must be analyzed. The data usage for each instruction can be calculated using a disassembler [21], and from the dependencies a local data flow graph can be produced for each basic block. (See Figure 1.)

$$\left\{\begin{matrix} CS: \\ DS: \\ SS: \\ ES: \\ FS: \\ GS: \end{matrix}\right\} \left[\left\{\begin{matrix} EAX \\ EBX \\ ECX \\ EDX \\ ESP \\ EBP \\ ESI \\ EDI \end{matrix}\right\}\right] + \left[\left\{\begin{matrix} EAX \\ EBX \\ ECX \\ EDX \\ EBP \\ ESI \\ EDI \end{matrix}\right\} * \left\{\begin{matrix} 1 \\ 2 \\ 4 \\ 8 \end{matrix}\right\}\right] + [\text{displacement}] \qquad (1)$$

The x86 instruction set in user mode uses 8 data registers (EAX, EBX, ECX, EDX, EBP, ESI, EDI, ESP), the instructions can access memory using a strict addressing mode (1). Due to the memory protection methods provided by modern operating systems, segment registers (CS, DS, SS, ES, FS, GS) are usually not used

in user space programs. It is safe to ignore those along with debug registers and control registers.

The x86 instruction set uses a special status flag register which contains the current state of the processor represented as different binary flags stored as bits of the register. These bits are used to store extra information about the last instruction and can be used to manipulate the outcome of the next instruction. For example the multiply instruction sets the overflow flag if the result cannot be represented on the current register due to overflow, the overflow flag can be tested in a conditional jump which may transfer execution to some error handling code.

Memory access uses a well defined formula but identifying whether two pointers are the same or not is impossible in the majority of the cases. For example [eax+14] , [00401000h] and [esp+14h] could point to the same address or to three completely different address. Some disassemblers, like Hex-Rays' IDA[1], can identify trivial cases for identical memory pointers but the analysis is complicated and should be done on the whole code section [21]. In our work we decided to treat the whole memory as one entity, the so called memory data. This simplifies the dependency model because every memory location can be treated as a single virtual register which can be used for both reading and writing. It is required to handle memory writes and reads because it will cause further data dependencies in the code.

We distinguish two kinds of data affection (read and write) and use 20 data types including registers: 8 basic x86 registers, 11 binary flags in the eflags register, and memory as a whole. Each instruction has a well defined data usage [16], which defines which registers / flags are used in different instructions, and how these registers affect the output. For simplicity each control flow instruction (such as condition, unconditional jump and call instructions) should be treated as if they write every data type. This ensures that during reordering within a basic block the control flow instructions remain at the end of the block.

The following rules are used to generate the data dependency :

A instruction should be before B instruction (considering original order A before B):

- if A reads any data type B writes, or
- if A writes any date type B writes, or
- if A writes any data type B reads.

The reordered and the original code have the same data flow graph, which means that they have the same effect (observable registers, flags and memory), only the control flow graph can be different, but at the end of each basic block the executed instructions have the same quantity, and only the execution order may vary [5, 12].

An example of basic block instruction data dependency is demonstrated in Figure 1. Each instruction is indicated by a separate box, the original position in the first line, and the actual instruction in the second line. The arrows represent data dependencies within the block. All the indirect dependencies are hidden. The circle marks the entry point of the basic block.

[1] http://www.hex-rays.com/products/ida/index.shtml (Last accessed: 2013-01-15)

4 Compression and permutation count

Most of the lossless data compression algorithms are designed to exploit statistical redundancy in the data. Reordering basic blocks could change the data statistics which may improve compression. Without any assumption about the actual data compression method we will consider a non-zero compression time which grows at least linearly with the size of the input data. To determine which input is the most compressible, the compression method should be executed on each input variation, then the one with the smallest compressed data size should be chosen.

The possible number of different reorderings for a basic block are bounded by $k!$, where k is the number of instructions within the block, but this number could be significantly lower because of the data flow constraints. Using the following formula the total permutation count can be calculated:

$$\prod_{i}^{N} \prod_{j}^{M_i} n_{i,j} \tag{2}$$

where N is the total number of functions, M_i is the number of basic blocks within the i^{th} function and $n_{i,j}$ is the permutation count for the j^{th} basic block in the i^{th} function. The best compression can be achieved only by testing all reorderings, which gives the global optimum. Due to the huge permutation count and the non-zero compression time this could be done only for very small files.

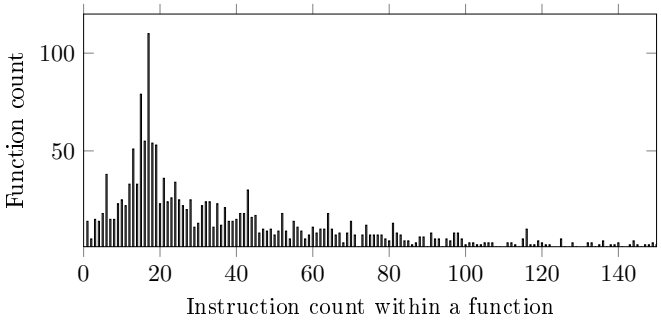

Figure 2: Function count by instruction count

Common programming methodology and practices suggest that the complexity (possible execution path count - branches) of each function should be relatively low. This way the source code can be easy to understand, maintain and test. During compilation the control flow can change after inlining/outlining functions but the distribution of the instruction count for functions does not change significantly. Analyzing a sample dataset which was taken from the libc system executable, a histogram can be created (Figure 2.), which shows the instruction count in functions. As expected, there are a lot of functions with only few instructions, the average function consists of 57 instructions, and only a couple of functions have more than 100 instructions.

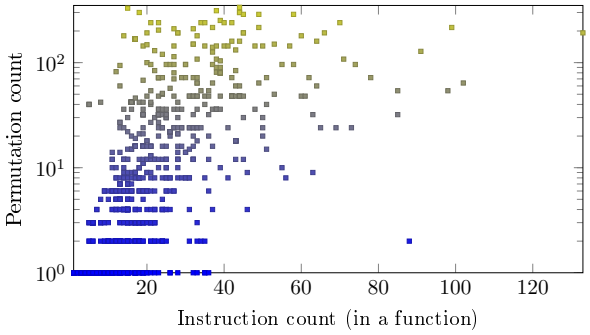

Figure 3: Permutation count

The permutation count (with valid data dependency) for each function was also calculated. The results can be seen on Figure 3. By increasing the instruction count, the permutation count of a function is also increasing. There is no common rule for the permutation count growth, low instruction count functions may also have a lot of permutations (e.g. "__strcspn_g" function in the dataset, 30 instructions, 76.951.350 possible permutations), and long functions may have only a few permutations (e.g. "getpass" function in the dataset, 133 instructions, 192 permutations). The permutation count heavily depends on efficient data type usage.

To keep the permutation count at a manageable level instead of searching for the global optimum in the compressibility local optimum places should be considered. The local optimum search is done on the function level, where each function is evaluated separately and every basic block permutation is tested within the function.The search algorithm is detailed in pseudo code in Algorithm 1.

first basic block perm. #1	second basic block perm. #1	third basic block perm. #1
first basic block perm. #2	second basic block perm. #1	third basic block perm. #1

...

first basic block perm. #K	second basic block perm. #1	third basic block perm. #1
first basic block perm. #1	second basic block perm. #2	third basic block perm. #1

...

Figure 4: Local optimum search iterations

This means that the optimal reordering for a selected function is calculated by compressing all the possible reorderings on the function level. In each iteration one

Algorithm 1 Pseudo code for the global suboptimum search

```
result=empty
for each function do
  perms=calculate every permutation of basic blocks in the function
  for each perms do
    compress the selected permutation
    if this is the smallest compressed size then
      incode=the uncompressed function
    end if
  end for
  append incode to result
end for
```

of the basic blocks gets another permutation and the whole function is recompressed and tested. As shown on Figure 4., this can be done with a simple limited counter function. This way the optimum search problem can be calculated in a distributed way, the algorithm should not have any information regarding the jump/call target for the control flow instructions.

In a basic block there is only one (if any) control flow instruction. This instruction is always at the end of a basic block. Among x86 instructions only control flow instructions have relative to current address pointers, that is why reordering instructions can be done by simply changing the instruction's order.

5 Implementation

In our implementation the function splitting is done by starting from the entry point and dynamic library export addresses then tracing the code using a disassembler and following each control flow edge with a depth-first-search. On each call instruction target a new function splitting point must be created. This way most of the code section gets analyzed, and using function split points the code can be split into functions. This result is double checked with the constraint for functions introduced in the second section, and if needed, the result gets modified.

Splitting functions into basic blocks are done by identifying the control flow instructions within a function, then splitting the code after the control flow instructions and finally at the point before the jump target points. This way the constraint for basic blocks can be granted (an example is shown in Figure 5., where 9 basic blocks have been identified in the example function). According to data flow analysis among these basic blocks only two can be reordered at the instruction level. The stack is heavily used to store data, which results in a large number of memory writes and reads.

The data dependency analysis is done using the free open source x86 disassembler called BeaEngine[2]. A binary dependency matrix gets defined which defines the

[2]http://www.beaengine.org/ (Last accessed: 2013-01-15)

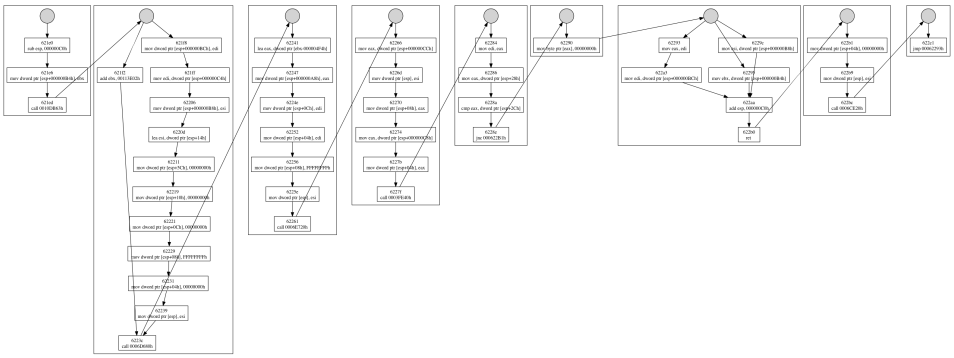

Figure 5: Data flow and basic blocks in a function

instruction order constraints. In the matrix the item $n_{i,j}$ is *true* if the i^{th} instruction within the basic block must be before the j^{th}. This constraint is transitive, so by propagating the items in the matrix, the transitive closure can be generated for easier access of the rules.

Some functions can still have a lot of permutations therefore only functions with less than 2.000 possible permutations were tested due to the computational limits. In the example shown on Figure 5. the 2nd block has 11 different permutations, the 7th block has 12 different permutations, so overall 132 cases have been tested.

The latest stable gzip[3] (version 1.3.12) and lzma[4] (version 5.0.3) software were used to compress the produced code. The parameters used for compression are:

- gzip: -9 (best compression)
- lzma: -e (extreme compression)

6 Results

For the sample function shown on Figure 5. the original code was 195 bytes compressed with gzip and 176 bytes compressed with lzma. Using reordering the compressed size was decreased by 2 bytes using gzip and 3 bytes using lzma, so the original code was not optimally ordered for compression purposes as shown on Figure 6.

To evaluate the explained method and the implementation, several files were tested from various compilers and operating systems. The filenames, operating systems, compilers and sources for the testfiles are shown in Table 2. In case of node.js, due to the high number of functions, only the first 1000 were tested. These binaries are commonly used on each operating system. All of these programs / libraries are written on a high level language (usually C++), which means that a

[3]http://www.gzip.org/ (Last accessed: 2013-01-15)
[4]http://tukaani.org/xz/ (Last accessed: 2013-01-15)

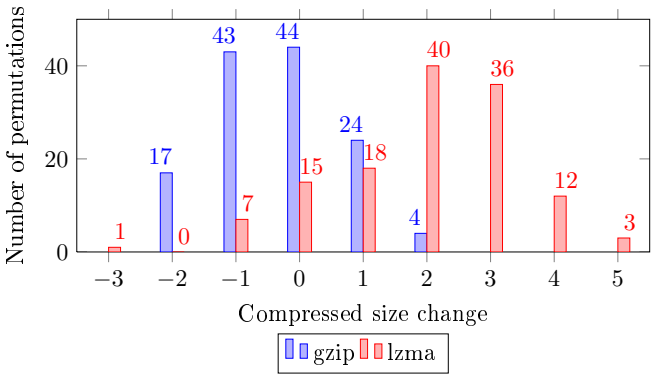

Figure 6: gzip and lzma compression results

Name	OS	Compiler	Source (Last accessed: 2013-01-15)
libc-2.13.so (-0ubuntu13.1)	Ubuntu	gcc	http://packages.ubuntu.com/natty/libc6-i386
unzip (6.0-4)	Debian	gcc	http://packages.debian.org/squeeze/unzip
libconfig.dll (1.4.8)	Windows	VS2008	http://www.hyperrealm.com/libconfig/
node.js (0.8.8)	Mac	llvm	http://nodejs.org/dist/latest/node-v0.8.8-darwin-x86.tar.gz

Table 2: Source of the files used in tests

compiler generates and optimizes the executable part of the binary. This excludes macro optimization which usually done at the machine code level by a human expert.

For verifying the statement, that the reordered code is functionally equal to the original, we performed several unit tests on the reordered functions. During the tests there were no major performance regressions which may arise due to reordering of code which was optimized for speed.

Name	Code section		Compressed size in bytes				Gain	
	in bytes		without reordering		with reordering		percentage	
	all	processed	gzip	lzma	gzip	lzma	gzip	lzma
libc-2.13.so	413.619	110.944	46.353	39.848	45.778	39.576	1.24%	0.68%
unzip	74.905	5.012	2.933	2.944	2.903	2.924	1.02%	0.67%
libconfig.dll	17.123	8.982	4.128	3.952	4.127	3.944	0.02%	0.20%
node.js	303.248	93.544	39.554	33.688	39.451	33.616	0.26%	0.21%

Table 3: Compression results

For evaluating the compression result, the non-machine code part of each file have to be omitted, the result should be compared using the reordered and the original data in compressed form. In Table 3. the result for compression tests can be seen, the function level statistics are in Table 4. Using the reordering method

Name	Method	Byte gain per function	
		Avg.	Std. dev.
libc-2.13.so	gzip	2.666	3.663
libc-2.13.so	lzma	0.993	2.094
unzip	gzip	1.288	1.766
unzip	lzma	1.244	2.227
libconfig.dll	gzip	0.583	1.096
libconfig.dll	lzma	0.366	1.269
node.js	gzip	0.902	1.650
node.js	lzma	0.543	1.604

Table 4: Detailed compression results

better results can be achieved: the compression size gain for gzip can be as high as 1.24%, for lzma 0.68%.

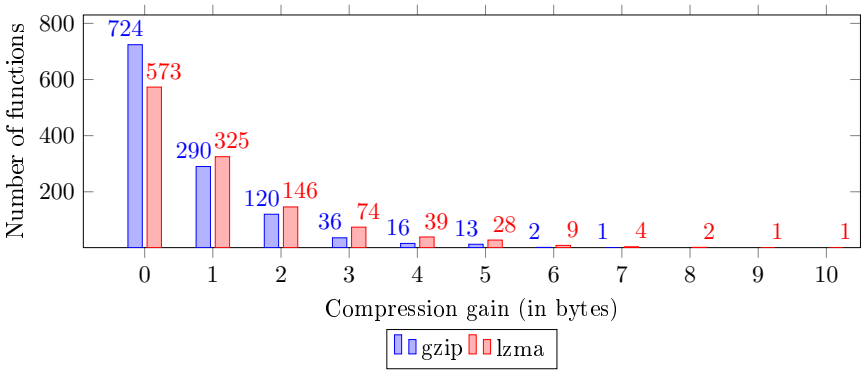

Figure 7: Compressed code size change

A detailed statistics on compression gain can be seen on Figure 7, these results are for libc-2.13.so. In this case more than 40% of all the functions can have a better compressible reordering than the original one but the compression gain is usually small (1-3 bytes/function) but significantly bigger gains are also possible (5-10 bytes / function).

7 Future work

Future work will include evaluation of other code transformation methods used together with instruction reordering, such as pattern based instruction substitution [14, 4, 17] and also exploiting the basic principles of the split-stream compression. The data types used in this work can be also refined using much more sophisticated analysis on memory access, especially on the stack. Instead of local

optimum search on function level, other search methods or optimum criteria should be considered, such as genetic algorithms.

References

[1] Linux kernel 2.6.30 changelog. Online. Last accessed: 2013-01-15. http://kernelnewbies.org/Linux_2_6_30.

[2] Beszédes, Árpád, Ferenc, Rudolf, Gyimóthy, Tibor, Dolenc, André, and Karsisto, Konsta. Survey of code-size reduction methods. *ACM Comput. Surv.*, 35(3):223–267, September 2003.

[3] Bruening, Derek L. Efficient, transparent and comprehensive runtime code manipulation, 2004.

[4] Bruschi, Danilo, Martignoni, Lorenzo, and Monga, Mattia. Code normalization for self-mutating malware. *IEEE Security and Privacy*, 5(2):46–54, March 2007.

[5] Chanet, Dominique, De Sutter, Bjorn, De Bus, Bruno, Van Put, Ludo, and De Bosschere, Koen. Automated reduction of the memory footprint of the linux kernel. *ACM Trans. Embed. Comput. Syst.*, 6(4), September 2007.

[6] Corporation, Microsoft. Microsoft portable executable and common object file format specification. Online. Last accessed: 2013-01-15. http://msdn.microsoft.com/en-us/library/windows/hardware/gg463119.aspx.

[7] Dai, Wenrui, Xiong, Hongkai, and Song, Li. On non-sequential context modeling with application to executable data compression. In *Data Compression Conference, 2008. DCC 2008*, pages 172 –181, march 2008.

[8] De Sutter, Bjorn, De Bus, Bruno, and De Bosschere, Koen. Sifting out the mud: low level c++ code reuse. *SIGPLAN Not.*, 37(11):275–291, November 2002.

[9] De Sutter, Bjorn, De Bus, Bruno, and De Bosschere, Koen. Link-time binary rewriting techniques for program compaction. *ACM Trans. Program. Lang. Syst.*, 27(5):882–945, September 2005.

[10] De Sutter, Bjorn, Van Put, Ludo, Chanet, Dominique, De Bus, Bruno, and De Bosschere, Koen. Link-time compaction and optimization of arm executables. *ACM Trans. Embed. Comput. Syst.*, 6(1), February 2007.

[11] Debray, Saumya K., Evans, William, Muth, Robert, and De Sutter, Bjorn. Compiler techniques for code compaction. *ACM Trans. Program. Lang. Syst.*, 22(2):378–415, March 2000.

[12] Drinić, Milenko, Kirovski, Darko, and Vo, Hoi. Ppmexe: Program compression. *ACM Trans. Program. Lang. Syst.*, 29(1), January 2007.

[13] He, Haifeng, Trimble, John, Perianayagam, Somu, Debray, Saumya, and Andrews, Gregory. Code compaction of an operating system kernel. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '07, pages 283–298, Washington, DC, USA, 2007. IEEE Computer Society.

[14] Hundt, Robert, Raman, Easwaran, Thuresson, Martin, and Vachharajani, Neil. Mao – an extensible micro-architectural optimizer. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '11, pages 1–10, Washington, DC, USA, 2011. IEEE Computer Society.

[15] Inc., Apple. Os x abi mach-o file format reference. Online. Last accessed: 2013-01-15. https://developer.apple.com/library/mac/#documentation /DeveloperTools/Conceptual/MachORuntime/ Reference/reference.html.

[16] Intel. Intel 64 and ia-32 architectures software developer manuals. Online. Last accessed: 2013-01-15. http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html.

[17] Kumar, Rajeev, Gupta, Amit, Pankaj, B. S., Ghosh, Mrinmoy, and Chakrabarti, P. P. Post-compilation optimization for multiple gains with pattern matching. *SIGPLAN Not.*, 40(12):14–23, December 2005.

[18] LIU Xian-Hua, YANG Yang, ZHANG Ji-Yu CHENG Xu. A basic-block reordering algorithm based on structural analysis. *Journal of Software*, 2008/19:1603–1612, 2008.

[19] Lucco, Steven. Split-stream dictionary program compression. *SIGPLAN Not.*, 35(5):27–34, May 2000.

[20] Morrill, Dan. Inside the android application framework. 2008.

[21] Paleari, Roberto, Martignoni, Lorenzo, Fresi Roglia, Giampaolo, and Bruschi, Danilo. N-version disassembly: differential testing of x86 disassemblers. In *Proceedings of the 19th international symposium on Software testing and analysis*, ISSTA '10, pages 265–274, New York, NY, USA, 2010. ACM.

[22] Pugh, William. Compressing java class files. *SIGPLAN Not.*, 34(5):247–258, May 1999.

[23] Schwarz, Benjamin, Debray, Saumya, Andrews, Gregory, and Legendre, Matthew. Plto: A link-time optimizer for the intel ia-32 architecture. In *In Proc. 2001 Workshop on Binary Translation (WBT-2001*, 2001.

[24] Sutter, Bjorn De, Bus, Bruno De, Bosschere, Koen De, and Debray, Saumya. Combining global code and data compaction. Technical report, 2001.